Mandrake: A Tool for Reverse-Engineering IBM Assembly Code

Paul Morris
Robert Filman

Software Technology Center
Lockheed Martin Missiles & Space
3251 Hanover Street O/H1-41 B/254G
Palo Alto, California 94304
{Morris|Filman}@stc.lockheed.com

Abstract

Assembly language code provides both a daunting challenge and a sterling opportunity for software reengineering. Ideally, we would like a system which, when fed assembly code, would automatically produce quality, maintainable high-level programs. That ideal, if not impossible, is certainly far beyond current technology. However, automation can profitably be applied to part of the task of reverse-engineering assembly code, by producing a "draft" of a high-level language version, to be verified, modified and polished by competent software reengineers. In this paper we describe our progress on developing a reengineer's apprentice to aid reverse-engineering of handwritten IBM 370 Assembly Code by automatically translating it to a higher-level form. This paper explores the problems that arise and some potential solutions, and describes the implementation of Mandrake, a system that performs an interesting collection of the reengineer's apprentice tasks.

1. Introduction

Many large software systems developed several decades ago are still in use today. These systems were built using platforms and languages that are increasingly obsolete. Their owners would often like to upgrade these applications to modern workstations and current languages, but are restrained by the cost of rewriting applications. This problem is particularly acute for programs developed in assembly language, which lack even a pretense of portability. In general, we would like to preserve the fruits of years of requirements analysis, design, and debugging, but typically this knowledge is represented solely in the source code. Manually translating such code is tedious and expensive. Our goal here is to automatically *levitate* such low-level code to a higher level, thereby simplifying software renovation. We recognize that quality, maintainable code will not be the product of such a transformation, but believe that producing quality high-level code can be greatly facilitated by a program that does much of the drudge work.

In this paper we describe Mandrake, a reengineer's apprentice that translates IBM 370 assembler code to a more compact and readable high-level form. Our goal is translated code that approaches the quality of natively-written high-level code, rather than simply being a Turing-equivalent emulation. We work under the assumption that we're dealing with handwritten assembler (as opposed to assembly code that was originally produced by a compiler [1,2]) and is in the same spirit as the Maintainer's Assistant [3]. In contrast with the Maintainer's Assistant, we are more concerned with the heuristic inference of almost-always true transformations than in pure semantics preservation. Our task is to infer the implicit patterns and idioms of human coding, not the rules of a compiler.

We note that an assembly programmer can take great liberties. Implicit overlays, casts, spaghetti control structures, type muddles and dispersing coherent computation to different parts of the program are common faults. To make the task tractable, we must recognize our limitations: we're not going to be able to successfully translate every legal assembly-code source program. We assume that programs are written in a relatively disciplined style, avoiding "pathologies." Second, the high-level translations are intended for human consumption, as an aid to understanding the assembly source, rather than for indiscriminate execution. Absolute correctness, while desirable, is to be subordinated to readability.

2. Overview

Our overall levitation process can be understood as having three major steps:

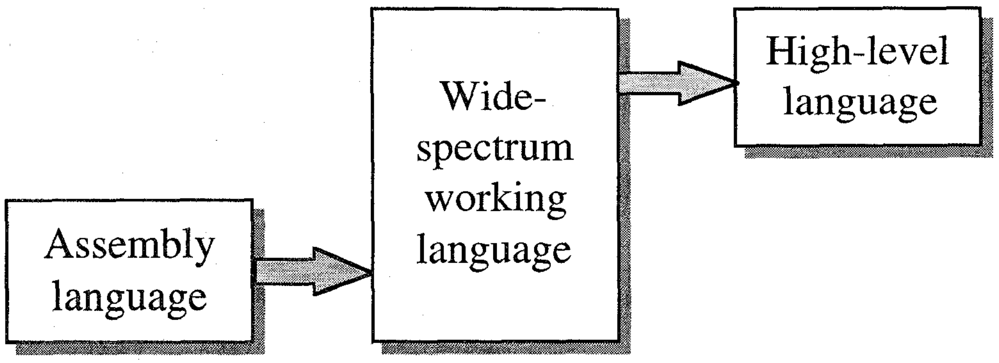

Figure 1. The steps for levitation

- An initial modular translation to a wide-spectrum *working language.*
- A set of transformations that progressively replace low-level constructs in the working language with higher-level forms.
- A translation from the higher-level form to the syntax of a particular target language.

These steps are illustrated in Figure 1.

By a *modular* translation, we mean one that is generally line-by-line. (This is not an absolute criterion because it is convenient to recognize and translate directly certain common multiline idioms.) A *wide-spectrum* language is one that provides both low-level machine-oriented constructs such as GOTOs and byte pointers, and higher level structures like if-statements, while-loops and arrays. C is an example of a wide-spectrum language. The language we use is similar to C except it has a more uniform syntax and semantics, and some additional constructs. Translating from a wide-spectrum language to a particular target language requires mapping the constructs in the wide-spectrum language to target language concepts and generating the appropriate syntax. The difficulty of this task varies by the flexibility of the target. Mandrake currently performs the first two steps. Thus, Mandrake is a system that reads in IBM assembler and produces pseudo-code that looks a lot like C.

2.1. Idiosyncrasies of assembler

Levitating assembler (more specifically, IBM 370 assembler) requires dealing with many idiosyncrasies. Some of the more critical ones for producing correct, maintainable code are

- **Data continuity.** Assembler allows the programmer to conceptualize a single data space which is flagged by certain names. It is common in assembler to provide multiple aliases for particular storage, based on their offset to different names. Good, maintainable high-level code (GMHLC) segments data into explicit records and arrays without regard to the arrangement of these elements with respect to each other.
- **Initialization.** Assembler provides a number of different ways to initialize storage, often based on the programmer's knowledge of data encodings or unusual repetition operators. GMHLC initializes data with respect to the constants of its data type; typical high-level languages lack complex, iterative initialization operators.
- **Addressing.** Assembler addressing is usually in terms of bytes. Thus, the fifth element of an array of twelve byte records is offset 12*(5-1)=48 from the base. GMHLC addressing is in terms of types. The fifth element of an array of anything is index 5.
- **Grain size.** Assembly code is *fine-grained.* Relatively little is accomplished by a single statement. With high-level languages, a single statement may evaluate a complex mathematical expression or (including its substatements) perform a complex iterative computation.
- **Control structures.** The only control structures in assembler are varieties of GOTOs. GMHLC uses more expressive iteration, conditional, error-handling and subprogram mechanisms.
- **Character encoding.** IBM assembler usually uses EBCDIC, a somewhat quirkier encoding than ASCII.
- **Conditionals.** Assemblers incorporate conditional branching in a two-step progress, first by setting a *condition code*, then (at perhaps a textually and computationally remote point) branching with respect to the condition code. GMHLC uses conceptually coherent, high-level constructs like **if** and **case** statements.
- **Precision.** In assembly language, numerical precision is maintained by the programmer, who must maneu-

ver through a maze of half, full and double word operations, and simple instructions that produce multi-register answers. GMHLC declares the types of data and allows the compiler to maintain precision, use the appropriate operators, and automatically perform necessary casts.

2.2. Pathologies

These differences give rise to a number of *pathologies* that make translation difficult. Clearly, extreme pathologies (like self-modifying code) are outside the range of this system. However, other pathologies are common, and are pathological only when misused. They cannot be ignored. One such pathology is the potential of assembly language data references to range over the entire space of data. This is problematical because the correctness of certain transformations depends on tracking set/use dependencies through the code. Since an assembly language reference can point to an arbitrary data location, such dependencies are in general undecidable. (For example, a section of code that ostensibly resets an element in an array might use an out-of-bounds reference to reset an arbitrary memory location, even a location in the execution-path of the program.) Adopting the conservative position that every *use* reference is dependent on every *set* reference is impractical, as then few transformations would ever be legal. One possibility is to use a theorem-prover to try to restrict the dependencies in particular instances, but that was not considered practical within the context of this project. Instead, we assume that programmer discipline excludes out-of-bound references—that is, if a offset reference is made with respect to an array base, that the value itself lies within the array.

Another issue is one that applies to any translation task. The source programming language may conflate concepts that are distinguished in the target language. For example, in C the null pointer, false and the integer zero can all be freely represented as 0. This poses problems when translating to languages that distinguish these concepts (e.g., Ada.) High-quality translation demands the contextually appropriate constant. In the case of IBM assembler, for example, the datum definitions `DC C'a'` (data constant character 'a') and `DC X'81'` (data constant hexadecimal '81') are equivalent, and assemble into the same machine code. However, the usages are different. If a translator respects the usage, as it should, the first form will be translated as a character, and the second as a number. On the other hand, a poorly-written assembly program might not follow this convention. This has the consequence that the validity of the translation may depend on the stylistic quality of the assembly source.

Because of these caveats, it is inadvisable to rely on the correctness of the translation. Mandrake is envisaged as an aid to human understanding of the assembly code rather than an autonomous translator. (In a porting situation, the output should be understood as a first draft to be manually checked, rather than the final translation.)

Having presented the above disclaimers, we add that the practical goal of the project was to accurately translate programs of the stylistic quality found in case studies contained in a text (e.g., [4]) on IBM assembler programming.

2.3. Overview of levitation

The operation of Mandrake can be divided into four steps, which cover the first two stages of the overall levitation process above:

[1.] Initial translation into primitive (low-level) working language code.

[2.1] Determining the control-flow and data-flow (set/use dependencies).

[2.2] Simplification and redesign of variables.

[2.3] GOTO elimination and procedure introduction.

We describe each phase in greater detail in subsequent sections. We have attempted to make the exposition self-contained so that it does not require a knowledge of IBM 370 assembly code. Also, the translations to working language that appear in examples have been written in an easily understood pseudocode form.

IBM 370 Assembly Language has a large variety of instructions and features. Mandrake's coverage is incomplete. Later in the paper, we will summarize the omitted features and discuss what is needed to include them.

Mandrake is implemented in Reasoning Systems' Software Refinery [5]. This system provides support for building systems to manipulate programs, including parsers to convert a text program into an abstract syntax tree (AST), the ability to decorate the nodes of such trees with additional information, and a rule system that facilitates pattern-based processing and transformation of the AST structures. Mandrake is one tool developed as part of Lockheed Martin's InVision project to aid software renovation [6].

3. Initial translation

Before the actual translation to working language code, there is a "preparation phase" that modifies the assembly code to make it easier to translate. These steps are shown in Figure 2. The preparation phase is needed because IBM 370 assembly language provides a variety of labor-saving ways for programmers to specify byte-counts and other values that are known or computable at assembly-time, but are tedious to express explicitly. For example, the expression `L'*` indicates the length in bytes of the machine-code form of the instruction in which it occurs. The translator, on the other hand, is better at dealing with explicit numeric values.

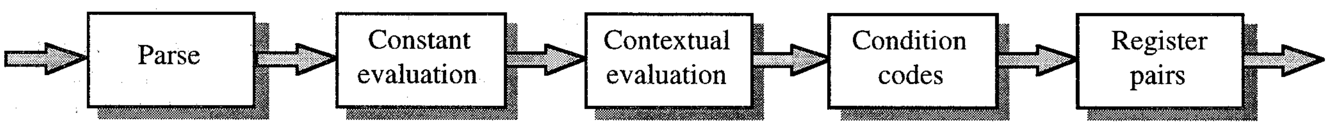

Figure 2. Initial translation steps

The preparation phase also makes explicit (by decorating the code) certain implicit or contextual information. For example, the assembly code may contain indirect references to data that rely on the contiguity of memory:

```
          PUT PRINTER RESULT
          ...
RESULT    DC ' THE DAY IS: '
W         DS 10C
          ...
PRINTER   DCB ... LRECL=23 ...
```

Here the PUT statement prints not only the RESULT string but also W! This is because the record length (23) for the printer specification reaches beyond the byte length (13) of RESULT to include the byte length (10) of W. The translation of the PUT instruction is facilitated by gathering symbol-table information, so that the record length and element structures are easily accessible from the PUT statement.

370 assembler allows greater flexibility in the initialization of data areas than is usually available in high-level languages. For example, (the functional equivalent of) a string may be initialized in a way that sets different character regions to varying values (including undefined). We have included similar flexibility in the working language. This preserves the modularity of the initial translation. (Later transformations could replace these unusual forms with procedural initialization routines, but this has not been implemented.)

The translation to primitive working language code has a principal phase that proceeds in postorder relative to the AST. Thus, the arguments of instructions are translated before the instructions themselves. However, contextual information is needed, since identically-appearing expressions may translate differently depending on the argument type, which in turn depends on the instruction type. Similarly, idioms are translated first to preempt the normal translations.

Two special issues arise for 370 assembly code: *condition codes*, and *even-odd register pairs*. The condition code is used to facilitate branching (changes in the flow of control). This is actually a two-bit quantity in the process status word, but may be more usefully thought of as a set of four mutually-exclusive global Boolean flags whose values are set when a comparison statement is executed. (Many arithmetic statements also set the condition code to record certain implicit comparisons.) The meanings of the global flags vary according to the type of statement, but generally represent the conditions x = y, x < y, and x > y for relevant values of x and y, and suitable (type-specific) interpretations of the comparison relations. The fourth flag is used to indicate errors. Branch statements may be conditioned on arbitrary disjunctions of the flags.

We have designed the translation to primitive working language code to introduce additional statements that explicitly set four global Boolean variables corresponding to the condition code flags. This expands the volume of code. Subsequent simplifying transformations based on data-flow analysis (discussed later) eliminate these globals. For example, the sequence

```
CR 3,4
BC 13,FOO
```

compares (as integers) the contents of registers 3 and 4, and then branches on a not-greater-than condition to FOO. This initially translates to the verbose

```
CC0 ← (R3 = R4);
CC1 ← (R3 < R4);
CC2 ← (R3 > R4);
CC3 ← false;
if (CC0 or CC1 or CC3) then
    goto FOO;
endif;
```

which is eventually simplifies to

```
if (R3 <= R4) then
    goto FOO;
endif;
```

This theme of initial expansion followed by contraction also occurs in the handling of so-called even-odd register pairs. This is a mechanism used in 370 assembly language for double-precision integers. Although double-precision is infrequently used, it is built into the multiplication and division instructions. This requires an initial complex translation for these operations. The translation can later be simplified subject to reasonable assumptions on the ranges of the inputs and outputs. The system attempts to prove the validity of these assumptions from the context of the code. If unsuccessful, it assumes them to be true anyway, and includes assertions to that effect in the translation. This allows a human expert to make common sense decisions regarding their validity. In some cases, the assertions may become runtime checks or comments in the reengineered code.

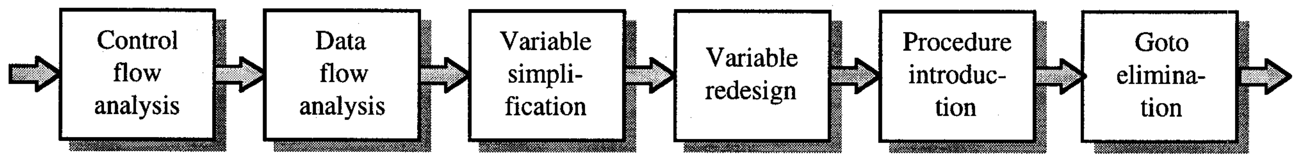

Figure 3. Working Language Manipulations

4. Working Language Manipulation

Having translated the assembler into the wide spectrum working language, the system proceeds to analyze the working language program and simplify it. The major steps of this analysis and simplification are illustrated in Figure 3.

4.1. Control-flow and data-flow

The code-improvement transformations described in subsequent sections require information about the flow of control and data within the program. This is computed in a preliminary information-gathering phase.

Our control-flow graph algorithm is relatively straightforward and unremarkable. One potential problem is the possibility of computed GOTOs, where the destination of a branch instruction may vary at runtime. In practice, computed GOTOs are rare except for returns from internal subroutines. In that case, the subroutine linkage is generally coded in a stereotyped way, which makes it possible to determine a restricted set of return locations. Arcs for each of these are included in the flowgraph. (Note that the occurrence of an arc indicates a *possible* transition, not a mandatory one. This is another case where a workable translation may depend on a non-pathological coding style in the assembly source.) Information about the return location of an internal subroutine must also be saved at the calling location, for use by later transformations that introduce procedure calls.

By data-flow analysis, we mean construction of the chains of set/use dependencies in the code. For reasons discussed above, dependency analysis is problematical. However, in practice it is rarely an issue, except in code that dereferences complex pointer structures.

The set/use computation involves determining a restricted collection of *set* locations that can affect the value of a variable at a particular *use* location. A standard algorithm [7] is used to propagate lists of setters down the control-flow graph to link up with users that depend upon them.

We have noticed some cases in which the downward propagation is inadequate for our purposes because of lost information about setters outside of (and prior to) the code being analyzed. Consider, for example, an external subroutine that splits into two branches after being started. Register 4 is set on one branch, but is not mentioned on the other. Suppose the two branches merge again, and then register 4 is used. According to the downward propagation, R4 seems to have exactly one setter, but in fact this register could have been set prior to entry into this segment of code, and control could have passed through the branch in which it was not reset. Missing this possibility could lead to certain incorrect simplifications of the code. Although in our experiments this difficulty has not arisen for subroutines, it would be a pressing issue if the system were to analyze arbitrary segments of code in isolation from the rest of the program. One way of rectifying this is to propagate imaginary “outside setters” for each use reference in the segment of code, but this seems unwieldy. A better alternative might be to complement the downward propagation with an upward propagation to determine the possible paths in which a use reference may be affected by a setter from outside. However, neither scheme has so far been implemented in Mandrake.

4.2. Variable simplification and redesign

As discussed above, the initial translation to working language code retains the fine-grained character of the assembler. Reducing this to a more concise form requires the elimination of intermediate variables.

Variable elimination is relatively straightforward once the set/use information is known. If a variable is set to an expression, and then used in only one place, and the expression is still live (its value has not been altered) at the time of use, then the expression is substituted for the variable, and the statement that sets the variable can be eliminated. (These changes generally require an update of the set/use information for the remaining variables, and of the control-flow graph.) Substitutions may also be performed in some cases that do not strictly conform to these conditions. For example, if the variable that is set also occurs in the set expression (e.g., `X ← X + 1`), then the expression is not currently live after the set statement, but may become live after the set statement is deleted so that a substitution is possible. After substitution has created complex algebraic expressions, Mandrake performs various algebraic simplifications by pattern-directed transformations.

Statements that set variables may be eliminated if the variables are not subsequently used. This process removes most condition code variables. Care must be taken to preserve variables that are visible outside the code being analyzed.

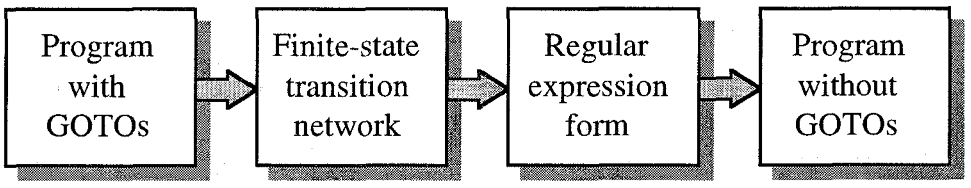

Figure 4. The GOTO-removal process

Mandrake then *redesigns* some of the remaining variables. For example, a segment of assembly code that processes an integer array will generally use an index register to step through the array. The register translates to an integer variable that determines which element of the array to work on next. Since all assembly language addresses are in bytes, the index is incremented by 4 to step through successive elements. However, high-level code uses word addresses rather than bytes, so that the index ought to be incremented by 1. This requires isolating a region of use of a variable, and superseding it with a redesigned variable whose value is 1/4 that of the original. More formally, we need to identify a collection S of settings of a variable, and a collection U of uses, such that every user of an element of S is in U, and every setter of an element of U is in S. Suppose this is done, and `X` is the variable. We introduce a new variable `x` such that `X = 4x`. Each setting `X ← e` in S is replaced by `x ← e/4`, and for each use in U, we substitute `4x` wherever `X` occurs. Notice that a single statement, say `X ← X+4`, may involve both a setting that is in S and a use that is in U. (This may happen when the statement is in a loop.) This poses no special problem. According to the above rules, the statement is transformed to `x ← (4x+4)/4`, which is simplified to `x ← x+1`.

The set/use information is also exploited to determine inputs, outputs, and local variables for subroutines, depending on whether first use is preceded by a set, and whether settings are visible after exit. (Typically, in IBM 370 assembler, old values are saved and restored.)

4.3. Procedure introduction and GOTO elimination

One of the critical issues in converting assembly code to a more readable form is the replacement of GOTOs or *jumps* by more structured statements. We may distinguish three basic types of jump: a returnless jump; a jump that saves a return location (similar to a *push-jump* in other assembly languages, but 370 assembly language does not utilize a stack); and a jump that returns to a saved location (similar to a *pop-jump*). We assume that programmer discipline ensures that the first two types of jump are to fixed locations in the control-flow graph, and the third type can be limited to a static set of easily-determinable return locations, one for each call.

Returnless jumps can be eliminated in favor of loops and conditionals. Interestingly, auxiliary variables are not needed, as long as nested loops with multilevel exits are allowed [8]. We have included a general loop construct with multilevel exits in the working language.

Previous work on removing GOTOs (e.g., [9]) has involved relatively ad-hoc methods. We have instead developed a general procedure based on the theory of finite automata. This makes use of the fact that a finite-state transition network can readily be converted to an equivalent regular expression. The approach includes a method for translating a computer program into a finite-state transition network, and a complex set of pattern-match operations to translate a resulting regular expression back into a computer program. This process is illustrated in Figure 4. Reference [10] describes this algorithm in more detail.

The removal of returnless jumps is algorithmic. We have assumed that programmer discipline ensures other kinds of jumps are only used to implement internal subroutines in a stereotyped manner. Mandrake uses pattern-directed transformations to replace returning jumps by high-level procedure calls and procedure definitions. This requires surgery on the control-flow graph. The control-flow successor at the subroutine call-point is redirected to the return location. The entry and exit of the subgraph corresponding to the subroutine definition are also redirected so that the definition will appear in an appropriate region of the graph.

The introduction of procedure definitions and procedure calls occurs before the conversion of the control-flow graph to code involving loops and conditionals. This allows the translator to correctly handle situations where the saved return location does not immediately follow the subroutine call-point in the assembly language text.

5. Coverage

In its current state, Mandrake handles most features of IBM 370 assembler including integer arithmetic, string operations, logic and comparisons, internal subroutines and jumps. However, several important elements are not yet covered. These include

- **Floating point arithmetic.** Floating point numbers are undoubtedly important. While integer arithmetic involves only register pairs, floating point arithmetic may utilize register pairs, triplets, and quartets. While Mandrake does not currently handle floating point, an extension to deal with floating point does not appear to involve new issues.

- **Interrupts and exceptions.** Interrupts, exceptions, and direct manipulations of the process status word appear to be beyond the reach of an automatic translation system at the present time, except perhaps for stereotyped usages. However, such translation may not be meaningful anyway in the context of reverse-engineering, since it is ill-advised for GMHLC to exploit exception mechanisms.
- **Packed/Zoned Decimals.** Packed/Zoned decimal arithmetic is an idiosyncratic feature of 370 assembler, providing decimal representations using strings of digits that can be manipulated directly by certain instructions. Packed/zoned decimal arithmetic appears to have arisen from impoverished I/O facilities. Some uses can be eliminated by idiom recognition, others uses can be translated to ordinary arithmetic and the remaining uses can be expressed as string-manipulations. Packed/Zoned arithmetic has a potential for expressing numbers with very high precision. Translation of such uses may have to be considered on a case-by-case basis, or may require custom translators.
- **Macros.** Macros are widely used in legacy code, and are the most important omission in the current system. There are two possible approaches to handling macros. The easier method is to expand before parsing. This could be done by a string-based preprocessor, or indeed by the 370 Assembler system itself. The drawback to this is that much of the higher-level structure is lost in the expansion. An alternative approach is to treat the macros as part of the language. However, IBM 370 assembler macros are meta-syntactic. For example, macro arguments may be concatenated as strings in the body of the macro definition *prior* to their interpretation as assembly language tokens, making them difficult to parse before macro expansion (because they may not obey the syntax of the underlying non-macro assembly language). On the other hand, an important use of macros in assembler is to provide a veneer of high-level control constructs within the framework of legal assembly code. In this case, it makes more sense to parse the macros as high-level units rather than following the normal assembly language syntax. These considerations suggest that macros be handled on a case-by-case, custom basis.
- **DSECTS.** DSECTS are generally used to pass complex record structures between assembly language program units. They are a significant omission in Mandrake. It is difficult to anticipate the issues that might arise in extending Mandrake in this direction, but there would be a substantial payoff, since recognition of record structures is an important aspect of reverse-engineering.

6. Examples

This section provides two examples of the performance of Mandrake, the first (Zeller's congruence) demonstrating the ability of the system to simplify arithmetic expressions and clichés, and the second illustrating the unraveling of a spaghetti control structure to loops.

6.1. Example: Zeller's congruence

The Zeller congruence is an algorithm for computing the day-of-week given the date. For example, the algorithm deduces that March 13, 1946 was a Wednesday. The algorithm uses a formula that is based on a modified Gregorian calendar where each year is considered to begin on March 1. (January and February are considered to belong to the previous year.)

Assuming "/" indicates integer division (the remainder is discarded), the formula gives the day-of-week number W (0 is Sunday, 1 is Monday, etc.) as

$$W = [(26M - 2)/10 + D + Y + Y/4 + C/4 - 2C] \bmod 7$$

where M is the month number, D is the day of the month, Y is the two-digit year within the century, and C is the century. All numbers are with respect to the modified calendar. Thus, for the 1946 date above, $M=1$, $D=13$, $Y=46$, and $C=19$.

Figure 5 shows an assembly language program to compute the day-of-week using the Zeller formula [4]. It takes as input a date in the normal Gregorian calendar, converts it to the modified calendar, and then applies the formula.

Figure 6 shows the same program after Mandrake has run. Note that the code is not only considerably simpler than the original but that font-size has grown, too.

6.2. Example: simplifying control structures

Our next example illustrates goto elimination in preference for higher-level loops. Mandrake levitates the assembly language matrix transposition program of Figure 7 (once again, from [4]) to the higher-level working language program of Figure 8. The goto's of this example form straightforward nested loops; Mandrake's GOTO elimination algorithm [10] works with arbitrarily complex spaghetti.

Besides these examples, the complete Mandrake system has been applied to other case studies found in an IBM 370 programming text [4]. In addition, part of the system, the GOTO removal facility [10], has been applied to industrial COBOL programs involving tens of thousands of lines of code [11]. The lack of coverage for macros has inhibited the application of the whole Mandrake system to large-scale examples.

```
ZELLER   CSECT
R2       EQU    2
         ...
R14      EQU    14
         STM    R14,R12,12(R13)
         BALR   R12,0
         USING  BASE,R12
BASE     ST     R13,SAVE+4
         LA     R13,SAVE
         OPEN   (READER,INPUT,PRINTER,OUTPUT)
         GET    READER,DATE
         PACK   TEMP(8),NUM1(10)
         CVB    R3,TEMP
         PACK   TEMP(8),NUM2(10)
         CVB    R5,TEMP
         ST     R5,DAY
         PACK   TEMP(8),NUM3(10)
         CVB    R5,TEMP
         S      R3,=F'2'
         BH     SKIP
         A      R3,=F'12'
         S      R5,=F'1'
SKIP     SR     R4,R4
         D      R4,=F'100'
         M      R2,=F'26'
         S      R3,=F'2'
         D      R2,=F'10'
         LR     R9,R3
         A      R9,DAY
         AR     R9,R4
         SR     R9,R5
         SR     R9,R5
         LR     R7,R4
         SR     R6,R6
         D      R6,=F'4'
         AR     R9,R7
         SR     R4,R4
         D      R4,=F'4'
         AR     R9,R5
         M      R8,=F'1'
         D      R8,=F'7'
         C      R8,=F'0'
         BNL    STORE
         A      R8,=F'7'
STORE    CVD    R8,TEMP
         OI     TEMP+7,X'0F'
         UNPK   W(10),TEMP(8)
         PUT    PRINTER,RESULT
ENDDATA  CLOSE  (READER,,PRINTER)
         L      R13,SAVE+4
         LM     R14,R12,12(R13)
         BR     R14
DATE     DS     80C
NUM1     EQU    DATE
NUM2     EQU    DATE+10
NUM3     EQU    DATE+20
DAY      DS     F
RESULT   DC     C' THE DAY IS: '
W        DS     10C
TEMP     DS     D
SAVE     DS     18F
READER   DCB    DSORG=PS,MACRF=GM,DDNAME=SYSIN,
                RECFM=F,LRECL=80,BLKSIZE=80,
                EODAD=ENDDATA
PRINTER  DCB
         DSORG=PS,MACRF=PM,DDNAME=SYSPRINT,
                RECFM=FA,LRECL=23,BLKSIZE=23
         END    ZELLER
```

Figure 5. Zeller's Congruence: Assembly Language

```
procedure ZELLER (in R14);
   locals R3, R5, R4, R2R3, R8;
   let DATE     = (CHAR [80] )?;
   let DAY      = (INT* )?;
   let W        = (CHAR [ 10] )?;
   let READER   =
      DEVICE (PS, GM, SYSIN, F, 80, 80);
   let PRINTER  =
      DEVICE  (PS, PM, SYSPRINT,
               FA, 23, 23);
   OPEN ( READER, INPUT,
          PRINTER, OUTPUT);
   GET ( READER, DATE);
   if (! EOF) then
   { SSCANF ( DATE, "10d", R3);
      SSCANF ( (DATE + 10), "10d",
               *(INT* ) DAY);
      SSCANF ( (DATE + 20), "10d", R5);
      set R3 = (R3 - 2);
      if (R3 ≤ 0) then
      {  set R3 = (R3 + 12);
         set R5 = (R5 - 1)
      }
      endif;
      assert (R5 ≥ 0);
      set R4 = (R5 rem 100);
      set R5 = (R5 / 100);
      set R2R3 = ((DOUBLE ) R3 * 26);
      assert (R2R3 ≤ MAXINT);
      set R8 =
         (((((((((R2R3 - 2) / 10) +
           *(INT* ) DAY) + R4) - R5) -
           R5) + (R4 / 4)) + (R5 / 4))
           rem 7);
      if (R8 < 0) then
      {
         set R8 = (R8 + 7)
      }
      endif;
      SPRINTF ( W, "10d", R8);
      PUT ( PRINTER,
         (CHAR* ) " THE DAY IS: ", W)}
   endif;
   CLOSE ( READER, PRINTER)
```

Figure 6. Zeller's Congruence: Translated Code

7. Related Work

Feldman and Friedman [12] describe the Bogart system for translating IBM assembly code. They identify many of the same issues, and adopt a similar flow analysis approach. However, Bogart produces code for direct execution rather than for human consumption, and readability is less of an issue. The paper focuses on a comparison with a pre-existing literal translator, and gives less emphasis to the detailed steps and the idiosyncrasies of 370 assembler. Bogart appears to differ from Mandrake in not doing general GOTO removal, or idiom recognition.

The approach of using a wide-spectrum working language was introduced by Ward [13]. Ward uses a formally-based language rather the C-like one considered here. This has been applied to the task of translating IBM assembler and other languages [14]. Transformations within the formal language are guaranteed correct. However, our experience suggests that "almost-always" correct interpretations are needed for readable translations. In [14], semantic approximations or "loose translations" are confined to the initial translation to the wide-spectrum language.

Andersen Consulting has developed the BAL/SRW Assembler re-engineering workbench [15]. Like Mandrake, this is a Refine-based system. The workbench contains an offline unit that is comparable to Mandrake, and an online unit that is more concerned with interactive browsing. Rather than translating to a separate language, BAL/SRW decorates the parsed assembly code with higher-level abstract structures. For example, it analyzes control flow, and recognizes stereotypical patterns. Unlike Mandrake, it does not appear to perform general GOTO removal.

8. Conclusions

We have described Mandrake, a system to aid in reengineering handwritten IBM 370 assembly code to quality, high-level code. The gap from assembly language to high-level is a large one, and compromises are neces-

```
*******************************************************************
** SUBPROGRAM TO TRANSPOSE AN N x N MATRIX IN PLACE.
** ASSUME N AND ARRAY ARE DEFINED
*******************************************************************
TRANS     ST   R14,TSAVE14      SAVE RETURN ADDRESS
          L    R3,N             GET N
          BCTR R3,0             N - 1
          LR   R10,R3           R10 = OUTER LOOP CONTROL = N - 1
          M    R2,=F'4'         R3 = CONSTANT = 4*(N -1)
          LA   R2,4(R3)         R2 = CONSTANT = 4*(N -1) + 4 = 4*N
          LA   R8,0             R8 = INDEX TO COLUMN ELEMENT = 0
          LA   R6,4             R6 = ADJ
* = NO. OF BYTES BETWEEN FIRST
* ELEMENT IN COLUMN AND ELEMENT
* IMMEDIATELY BELOW DIAGONAL
* REPEAT N - 1 TIMES:
TROUTLP   LR   R9,R10           R9 = INNER LOOP COUNTER L=N-1,...,1
          LA   R8,0(R8,R6)      R8 = INDEX TO COLUMN ELEMENT
* IMMEDIATELY BELOW DIAGONAL
          LA   R7,0(R8,R3)      R7 = INDEX TO ROW ELEMENT
* IMMEDIATELY RIGHT OF DIAGONAL
* - REPEAT L = C(R9) TIMES:
TRINLP    L    R4,ARRAY(R8)     GET COLUMN ELEMENT
          L    R5,ARRAY(R7)     GET ROW ELEMENT
          ST   R5,ARRAY(R8)     INTERCHANGE
          ST   R4,ARRAY(R7)     THEM
          LA   R8,4(R8)         POINT TO NEXT COLUMN ELT.
          LA   R7,0(R7,R2)      POINT TO NEXT ROW ELT.
          BCT  R9,TRINLP        DECR./REPEAT UNTIL ZERO
* - END REPEAT (R8 NOW POINTS TO TOP OF NEXT COLUMN)
          LA   R6,4(R6)         ADJ = ADJ + 4
          BCT  R10,TROUTLP      DECR./REPEAT UNTIL ZERO
* END REPEAT
          L    R14,TSAVE14      RESTORE RETURN ADDRESS
          BR   R14              RETURN
TSAVE14   DS   F                RETURN ADDRESS SAVE AREA
```

Figure 7. Matrix transposition: Assembly Code

```
procedure TRANS ();
   set *(INT* ) TSAVE14 = R14;
   set R3 = (*(INT* ) N - 1);
   set R10 = R3;
   assert (R3 ≤ (MAXINT / 4));
   set R804 = 0;
   set R604 = 1;
   loop / 1
   {  set R9 = R10;
      set R804 = (R804 + R604);
      set R704 = (R804 + R3);
      loop / 2
      {  set R4 = ARRAY [ R804];
         set R5 = ARRAY [ R704];
         set ARRAY [ R804] = R5;
         set ARRAY [ R704] = R4;
         set R804 = (R804 + 1);
         set R704 = (R704 + (R3 + 1));
         set R9 = (R9 - 1);
         if (R9 == 0) then
            {exit / 2}
         endif}
      endloop / 2;
      set R604 = (R604 + 1);
      set R10 = (R10 - 1);
      if (R10 == 0) then
         {exit / 1}
      endif}
   endloop / 1
end
```

Figure 8. Matrix transposition: working language translation

sary: the system assumes some degree of discipline of style in the assembly program input. This sufficed for the satisfactory translation of programs of the level of quality of case studies in a manual. Nevertheless, the issue of correctness is a pressing one, and would seem to require a long process of testing and evolution to achieve a robust translation system.

References

[1] P. Breuer, and J. Bowen. Decompilation: The Enumeration of Types and Grammars. *ACM Trans. on Prog. Lang. Sys.*, 16 (5):1613–1647, 1994.

[2] C. R. Hollander. *Decompilation of Object Programs.* Ph.D. thesis, Stanford University, 1973.

[3] K. Bennett, T. Bull, and H. Yang. A transformation system for maintenance—turning theory into practice. *Proc. Conf. on Software Maintenance 1992, IEEE*: 146–155.

[4] F. M. Carrano. *Assembler Language Programming for the IBM 370.* Benjamin/Cummings, Menlo Park, CA, 1988.

[5] Reasoning Systems. REFINE User's Guide, Version 3.0, Reasoning Systems Inc, 1990.

[6] R. E. Filman, Applying AI to software reengineering. to appear in *Automated Software Engineering*, 1996.

[7] W. A. Barrett, and J. D. Couch. *Compiler Construction Theory and Practice* Science Research Associates, Chicago, IL, 1979.

[8] W. W. Peterson, T. Kasami, and N. Tokura. On the Capabilities of While, Repeat, and Exit Statements. *Communications of the ACM* 16(8), August, 1973.

[9] E. Ashcroft and Z. Manna. The translation of 'goto' to 'while' programs. *Information Processing* 71, 1972.

[10] P. Morris, R. Grey, and R. Filman, GOTO Removal Based On Regular Expressions. Submitted, 1996.

[11] W. Polak, T. Bickmore, and L. Nelson. Reengineering IMS databases to Relational Systems. *Proc. Seventh Annual Software Technology Conference*, CD-ROM, 1995.

[12] Y.A. Feldman, and D.A. Friedman. Portability by Automatic Translation; A large-Scale Case Study. *Proc. 10th Knowledge-Based Software Engineering Conference*, 1995.

[13] M. Ward. *Program Analysis by Formal Transformation.* D.Phil. thesis, Oxford University, 1989.

[14] M. P. Ward and K. H. Bennett. *Formal Methods for Legacy Systems.* Dept. of Computer Science, U. of Durham, 1995.

[15] W. Kozaczynski, E. S. Liongosari, and J. Q. Ning. BAL/SRW: Assembler re-engineering workbench. *Information and Software Technology* 33(9), November 1991.